meshwalk-2.0*

Claude Heiland-Allen†

2018

```
/*
  meshwalk -- audiovisual drone
  Copyright (C) 2018  Claude Heiland-Allen <claude@mathr.co.uk>

  This program is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <https://www.gnu.org/licenses/>.
*/
/*
# build (or use `make`)
gcc -std=c99 -Wall -Wextra -pedantic -Ofast -ffast-math -march=native \
    -o meshwalk-2.0 meshwalk-2.0.c \
    -ljack -lsndfile -lGLEW -lGL -lglfw -lm

# realtime mode with JACK audio, press Q to exit
./meshwalk-2.0

# render
for i in $(seq 180) ; do pngtopnm < meshwalk-2.0-title.png ; done |
ffmpeg -r 60 -i - -vf fade=in:30:30,fade=out:120:30 meshwalk-2.0-title-%04d.ppm
for i in $(seq 180) ; do pngtopnm < meshwalk-2.0-credits.png ; done |
ffmpeg -r 60 -i - -vf fade=in:30:30,fade=out:120:30 ↙
    ↳ meshwalk-2.0-credits-%04d.ppm
(
  cat meshwalk-2.0-title-0*.ppm
  ./meshwalk-2.0 -nrt
  cat meshwalk-2.0-credits-0*.ppm
) |
ffmpeg -framerate 60 -i - -codec:v png meshwalk-2.0_v.mov
ffmpeg -i meshwalk-2.0_v.mov -i meshwalk-2.0_a.mov \
  -codec:v copy -codec:a copy -movflags +faststart \
  meshwalk-2.0.mov
ffmpeg -i meshwalk-2.0.mov \
  -pix_fmt yuv420p -profile:v high -level:v 4.1 -tune:v animation \
  -crf:v 20 -b:a 384k -movflags +faststart \
  meshwalk-2.0.mp4
*/

```

*https://mathr.co.uk/meshwalk/

†mailto:claude@mathr.co.uk

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include <GL/glew.h>
#include <GLFW/glfw3.h>

#include <jack/jack.h>
#include <sndfile.h>

#define SR 48000
#define FPS 60
#define W (16 * 4)
#define H ( 9 * 4)

#define pi 3.141592653589793

// precondition: 0 <= phase <= 1
static inline float cosf9(float phase) {
  float p = fabsf(4.0f * phase - 2.0f) - 1.0f;
  float p2 = p * p;
  // p in -1 .. 1
  float s
    = 1.5707963267948965580e+00f * p
    - 6.4596271553942852250e-01f * p * p2
    + 7.9685048314861006702e-02f * p * p2 * p2
    - 4.6672571910271187789e-03f * p * p2 * p2 * p2
    + 1.4859762069630022552e-04f * p * p2 * p2 * p2 * p2;
  // compiler figures out optimal simd multiplications
  return s;
}

static inline float wrap(float x) { return x - floor(x); }

static void debug_program(GLuint program) {
  GLint status = 0;
  glGetProgramiv(program, GL_LINK_STATUS, &status);
  GLint length = 0;
  glGetProgramiv(program, GL_INFO_LOG_LENGTH, &length);
  char *info = 0;
  if (length) {
    info = malloc(length + 1);
    info[0] = 0;
    glGetProgramInfoLog(program, length, 0, info);
    info[length] = 0;
  }
  if ((info && info[0]) || ! status) {
    fprintf(stderr, "program link info:\n%s", info ? info : "(no info log)");
  }
  if (info) {
    free(info);
  }
}

static void debug_shader(GLuint shader, GLenum type, const char *source) {
  GLint status = 0;
  glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
  GLint length = 0;
  glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &length);
  char *info = 0;
  if (length) {
```

```
108       info = malloc(length + 1);
109       info[0] = 0;
110       glGetShaderInfoLog(shader, length, 0, info);
111       info[length] = 0;
112     }
113     if ((info && info[0]) || ! status) {
114       const char *type_str = "unknown";
115       switch (type) {
116         case GL_VERTEX_SHADER: type_str = "vertex"; break;
117         case GL_FRAGMENT_SHADER: type_str = "fragment"; break;
118         case GL_COMPUTE_SHADER: type_str = "compute"; break;
119       }
120       fprintf
121         ( stderr
122         , "%s shader compile info:\n%s\nshader source:\n%s"
123         , type_str
124         , info ? info : "(no info log)"
125         , source ? source : "(no source)"
126         );
127     }
128     if (info) {
129       free(info);
130     }
131   }
132
133   static GLuint vertex_fragment_shader(const char *vert, const char *frag) {
134     GLuint program = glCreateProgram();
135     {
136       GLuint shader = glCreateShader(GL_VERTEX_SHADER);
137       glShaderSource(shader, 1, &vert, 0);
138       glCompileShader(shader);
139       debug_shader(shader, GL_VERTEX_SHADER, vert);
140       glAttachShader(program, shader);
141       glDeleteShader(shader);
142     }
143     {
144       GLuint shader = glCreateShader(GL_FRAGMENT_SHADER);
145       glShaderSource(shader, 1, &frag, 0);
146       glCompileShader(shader);
147       debug_shader(shader, GL_FRAGMENT_SHADER, frag);
148       glAttachShader(program, shader);
149       glDeleteShader(shader);
150     }
151     glLinkProgram(program);
152     debug_program(program);
153     return program;
154   }
155
156   // positions (even store average, odd store edge offsets)
157   static float g[H][W];
158   // velocities (odd for edges)
159   static float dg[H][W];
160
161   // OpenGL triangles: two quads per even coordinate, convexity guaranteed
162   static GLfloat t[H/2 + 2][W/2 + 2][2][2][3][3];
163
164   // audio oscillators: one per quad, double buffered with linear interpolation
165   static int a_which = 0;
166   static float pan  [2][H/2 + 2][W/2 + 2][2];
167   static float pitch[2][H/2 + 2][W/2 + 2][2];
168   static float level[2][H/2 + 2][W/2 + 2][2];
```

```
169 static float phase[H/2 + 2][W/2 + 2][2];
170
171 // http://burtleburtle.net/bob/hash/integer.html
172 static uint32_t burtle_hash(uint32_t a)
173 {
174     a = (a+0x7ed55d16) + (a<<12);
175     a = (a^0xc761c23c) ^ (a>>19);
176     a = (a+0x165667b1) + (a<<5);
177     a = (a+0xd3a2646c) ^ (a<<9);
178     a = (a+0xfd7046c5) + (a<<3);
179     a = (a^0xb55a4f09) ^ (a>>16);
180     return a;
181 }
182
183 // pseudo-random uniform number in [0,1)
184 static float uniform(uint32_t x, uint32_t y, uint32_t c)
185 {
186   return
187     burtle_hash(x +
188     burtle_hash(y +
189     burtle_hash(c))) /
190     (float) (0x100000000LL);
191 }
192
193 // randomize positions, no velocity
194 static void init()
195 {
196   for (int j = 0; j < H; ++j)
197   for (int i = 0; i < W; ++i)
198   {
199     g[j][i] = 0.25 + 0.5 * uniform(i, j, 0);
200     dg[j][i] = 0;
201   }
202   for (int j = 0; j < H/2 + 2; ++ j)
203   for (int i = 0; i < W/2 + 2; ++ i)
204   {
205     phase[j][i][0] = uniform(i, j, 1);
206     phase[j][i][1] = uniform(i, j, 2);
207   }
208 }
209
210 // compute average deviation from even coordinates
211 static void even()
212 {
213   for (int j = 0; j < H; ++j)
214   for (int i = 0; i < W; ++i)
215   if (((i & 1) == 0) && ((j & 1) == 0))
216   {
217     int ip = (i + 1) % W;
218     int jp = (j + 1) % H;
219     int im = (i - 1 + W) % W;
220     int jm = (j - 1 + H) % H;
221     float xp =      g[j][ip];
222     float xm = 1 - g[j][im];
223     float yp =      g[jp][i];
224     float ym = 1 - g[jm][i];
225     float m = 0.25f * (xp + xm + yp + ym);
226     g[j][i] = m;
227   }
228 }
229
```

```
230 // update deviation of even coordinates
231 static void odd(int k)
232 {
233   // integration time constant
234   const float dt = 1.0f/32.0f;
235   // friction proportional to velocity
236   const float friction = 0.07f * dt;
237   // elastic collisions at the end points of the intervals
238   const float elastic = 0.125f;
239   // time-varying forcing of the spring constant gives interesting effects
240   const float spring = 0.1f + 0.02f * sinf(2.0f * pi * k / 60.0f);
241
242   for (int j = 0; j < H; ++j)
243   for (int i = 0; i < W; ++i)
244   if (((i + j) & 1) == 1) // edges
245   {
246     // get current position
247     float a = g[j][i];
248     // get aveage of endpoints
249     float ap, am;
250     if (i & 1) // horizontal
251     {
252       int ip = (i + 1) % W;
253       int im = (i - 1 + W) % W;
254       ap = g[j][ip];
255       am = g[j][im];
256     }
257     else // vertical
258     {
259       int jm = (j - 1 + H) % H;
260       int jp = (j + 1) % H;
261       ap = g[jp][i];
262       am = g[jm][i];
263     }
264     float target = 0.5f * (ap + 1.0f - am);
265     // get current velocity
266     float d = dg[j][i];
267     // compute force
268     float f = spring * (target - a) - friction * d;
269     // integrate position
270     a = a + dt * d;
271     // handle far out of bounds
272     if (a >  2) { a = 0.5; d = 0; }
273     if (a < -1) { a = 0.5; d = 0; }
274     // handle elastic reflections at end points
275     if (a > 1) { a = 1 - (a - 1); d = - elastic * d; }
276     if (a < 0) { a = 0 - (a - 0); d = - elastic * d; }
277     // store new position
278     g[j][i] = a;
279     // integrate velocity
280     dg[j][i] = d + dt * f;
281   }
282 }
283
284 // update triangle positions for OpenGL
285 static void tris()
286 {
287   int w = 1 - a_which;
288   float base_pitch[4] = { 150.0f / SR, 250.0f / SR, 150.0f / SR, 100.0f / SR };
289   float gain = 0.5f;
290   float volume = gain / sqrtf((H/2 + 2) * (W/2 + 2) * 2);
```

```
291   for (int j0 = -1; j0 <= H; ++j0)
292   for (int i0 = -1; i0 <= W; ++i0)
293   {
294     int i = (i0 + W) % W;
295     int j = (j0 + H) % H;
296     if ((i & 1) == (j & 1)) // even coordinates own a quad
297     {
298       int x = i0 >> 1;
299       int y = j0 >> 1;
300       int j1 = y + 1;
301       int i1 = x + 1;
302       int f = j & 1; // quad type (coordinates both even vs both odd)
303       int c = f | ((i0 ^ j0) & 2); // colour
304       t[j1][i1][f][0][0][0] = x + f;
305       t[j1][i1][f][0][0][1] = y;
306       t[j1][i1][f][0][0][2] = c;
307       t[j1][i1][f][0][1][0] = x + 1;
308       t[j1][i1][f][0][1][1] = y + f;
309       t[j1][i1][f][0][1][2] = c;
310       t[j1][i1][f][0][2][0] = x + f;
311       t[j1][i1][f][0][2][1] = y + 1;
312       t[j1][i1][f][0][2][2] = c;
313       t[j1][i1][f][1][0][2] = c;
314       t[j1][i1][f][1][1][2] = c;
315       t[j1][i1][f][1][2][0] = x;
316       t[j1][i1][f][1][2][1] = y + f;
317       t[j1][i1][f][1][2][2] = c;
318       float top    = g[(j    ) % H][(i + 1) % W];
319       float right  = g[(j + 1) % H][(i + 2) % W];
320       float bottom = g[(j + 2) % H][(i + 1) % W];
321       float left   = g[(j + 1) % H][(i    ) % W];
322       float xpos0, ypos0;
323       (void) xpos0;
324       {
325         float x0 = t[j1][i1][f][0][0][0];
326         float x1 = t[j1][i1][f][0][1][0];
327         float x2 = t[j1][i1][f][0][2][0];
328         float x3 = t[j1][i1][f][1][2][0];
329         float y0 = t[j1][i1][f][0][0][1];
330         float y1 = t[j1][i1][f][0][1][1];
331         float y2 = t[j1][i1][f][0][2][1];
332         float y3 = t[j1][i1][f][1][2][1];
333         xpos0 = 0.25f * (x0 + x1 + x2 + x3 + 1.0f);
334         ypos0 = 0.25f * (y0 + y1 + y2 + y3 + 1.0f);
335       }
336       t[j1][i1][f][0][0][  f] += top;
337       t[j1][i1][f][0][1][1-f] += right;
338       t[j1][i1][f][0][2][  f] += bottom;
339       t[j1][i1][f][1][2][1-f] += left;
340       float x0 = t[j1][i1][f][0][0][0];
341       float x1 = t[j1][i1][f][0][1][0];
342       float x2 = t[j1][i1][f][0][2][0];
343       float x3 = t[j1][i1][f][1][2][0];
344       float y0 = t[j1][i1][f][0][0][1];
345       float y1 = t[j1][i1][f][0][1][1];
346       float y2 = t[j1][i1][f][0][2][1];
347       float y3 = t[j1][i1][f][1][2][1];
348       float xpos = 0.25f * (x0 + x1 + x2 + x3);
349       float ypos = 0.25f * (y0 + y1 + y2 + y3);
350       float d1x = x2 - x0;
351       float d2x = x3 - x1;
```

```
352         float d1y = y2 - y0;
353         float d2y = y3 - y1;
354         d1x *= d1x;
355         d2x *= d2x;
356         d1y *= d1y;
357         d2y *= d2y;
358         float d1 = d1x + d1y;
359         float d2 = d2x + d2y;
360         float areaSquared = d1 * d2;
361         t[j1][i1][f][1][0][0] = t[j1][i1][f][0][0][0];
362         t[j1][i1][f][1][0][1] = t[j1][i1][f][0][0][1];
363         t[j1][i1][f][1][1][0] = t[j1][i1][f][0][2][0];
364         t[j1][i1][f][1][1][1] = t[j1][i1][f][0][2][1];
365         pan  [w][j1][i1][f] = xpos / ((W >> 1) + 2) * 0.25f;
366         pitch[w][j1][i1][f] = pow(1.5, ypos - ypos0) * base_pitch[c];
367         level[w][j1][i1][f] = areaSquared * volume;
368       }
369     }
370   }
371
372   static void audiocb(float *out_left, float *out_right, int nframes)
373   {
374     for (int n = 0; n < nframes; ++n)
375     {
376       out_left[n] = out_right[n] = 0;
377     }
378     int w = a_which;
379     int w1 = 1 - w;
380     for (int j0 = -1; j0 <= H; ++j0)
381     {
382       float l[nframes], r[nframes];
383       for (int n = 0; n < nframes; ++n)
384       {
385         l[n] = r[n] = 0;
386       }
387       for (int i0 = -1; i0 <= W; ++i0)
388       {
389         int i = (i0 + W) % W;
390         int j = (j0 + H) % H;
391         if ((i & 1) == (j & 1)) // even coordinates own a quad
392         {
393           int x = i0 >> 1;
394           int y = j0 >> 1;
395           int j1 = y + 1;
396           int i1 = x + 1;
397           int f = j & 1; // quad type (coordinates both even vs both odd)
398           int c = f | ((i0 ^ j0) & 2); // colour
399           float pan_0c = cosf9(        pan[w ][j1][i1][f]);
400           float pan_0s = cosf9(0.25f - pan[w ][j1][i1][f]);
401           float pan_1c = cosf9(        pan[w1][j1][i1][f]);
402           float pan_1s = cosf9(0.25f - pan[w1][j1][i1][f]);
403           float pit_0 = pitch[w ][j1][i1][f];
404           float pit_1 = pitch[w1][j1][i1][f];
405           float lev_0 = level[w ][j1][i1][f];
406           float lev_1 = level[w1][j1][i1][f];
407           for (int n = 0; n < nframes; ++n)
408           {
409             float lin = (n + 0.5) / nframes;
410             float lin1 = 1 - lin;
411             float pan1c  = pan_0c* lin1 + pan_1c* lin;
412             float pan1s  = pan_0s* lin1 + pan_1s* lin;
```

```
413             float pitch1 = pit_0 * lin1 + pit_1 * lin;
414             float level1 = lev_0 * lin1 + lev_1 * lin;
415             float p = phase[j1][i1][f] = wrap(phase[j1][i1][f] + pitch1);
416             float o = 0;
417             switch (c)
418             {
419               case 0: o = 0.5f - p; break;
420               case 1: o = fabsf(p - 0.5f) - 0.25f; break;
421               case 2: o = p - 0.5f; break;
422               case 3: o = (p < 0.5f) - 0.5f; break;
423             }
424             o *= level1;
425             l[n] += o * pan1c;
426             r[n] += o * pan1s;
427           }
428         }
429       }
430       for (int n = 0; n < nframes; ++n)
431       {
432         out_left[n] += l[n];
433         out_right[n] += r[n];
434       }
435     }
436     a_which = 1 - a_which;
437   }
438
439   jack_client_t *client;
440   jack_port_t *port[2];
441   static int processcb(jack_nframes_t nframes, void *arg) {
442     (void) arg;
443     jack_default_audio_sample_t *out[2];
444     out[0] = jack_port_get_buffer(port[0], nframes);
445     out[1] = jack_port_get_buffer(port[1], nframes);
446     audiocb(out[0], out[1], nframes);
447     return 0;
448   }
449
450   // GLFW keyboard callback
451   static void keycb
452     ( GLFWwindow *window
453     , int key
454     , int scancode
455     , int action
456     , int mods
457     )
458   {
459     (void) key;
460     (void) scancode;
461     (void) mods;
462     if (action == GLFW_PRESS)
463       glfwSetWindowShouldClose(window, GL_TRUE);
464   }
465
466   // entry point
467   extern int main(int argc, char **argv)
468   {
469     (void) argv;
470     int RECORD = argc > 1;
471     // start OpenGL 3.3 core profile 1920x1080 full screen
472     int w = 1920;
473     int h = 1080;
```

```
474   glfwInit();
475   glfwWindowHint(GLFW_CLIENT_API, GLFW_OPENGL_API);
476   glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
477   glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
478   glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
479   glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
480   glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
481   glfwWindowHint(GLFW_DECORATED, GL_FALSE);
482   GLFWwindow *window = glfwCreateWindow(w, h, "meshwalk", 0, 0);
483   glfwMakeContextCurrent(window);
484   glfwSetKeyCallback(window, keycb);
485   glewExperimental = GL_TRUE;
486   glewInit();
487   glGetError(); // discard common error from glew
488   // set up vertex array object
489   glClearColor(0, 1, 0, 1);
490   GLuint vao;
491   glGenVertexArrays(1, &vao);
492   glBindVertexArray(vao);
493   // set up buffer
494   GLuint vbo;
495   glGenBuffers(1, &vbo);
496   glBindBuffer(GL_ARRAY_BUFFER, vbo);
497   glBufferData(GL_ARRAY_BUFFER, sizeof(t), 0, GL_DYNAMIC_DRAW);
498   // set up framebuffer
499   GLuint fbo;
500   glGenFramebuffers(1, &fbo);
501   glBindFramebuffer(GL_FRAMEBUFFER, fbo);
502   GLuint tex;
503   glGenTextures(1, &tex);
504   glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, tex);
505   glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE, 8, GL_RGB, w, h, GL_FALSE);
506   glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, ↙
        ↳ GL_TEXTURE_2D_MULTISAMPLE, tex, 0);
507   GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
508   if (status != GL_FRAMEBUFFER_COMPLETE)
509     fprintf(stderr, "GL_FRAMEBUFFER_STATUS_%d\n", status);
510   // set up shader
511   const char *s_vertex =
512     "#version 330 core\n"
513     "uniform mat4 MVP;\n"
514     "layout (location = 0) in vec3 position;\n"
515     "flat out int c;\n"
516     "void main()\n"
517     "{\n"
518     "  c = int(position.z);\n"
519     "  gl_Position = MVP * vec4(position.xy, 0.0, 1.0);\n"
520     "}\n"
521     ;
522   const char *s_fragment =
523     "#version 330 core\n"
524     "flat in int c;\n"
525     "layout (location = 0) out vec4 colour;\n"
526     "void main()\n"
527     "{\n"
528     "  vec3 blue = vec3(0.0, 0.5, 1.0);\n"
529     "  vec3 white = vec3(1.0);\n"
530     "  vec3 red = vec3(1.0, 0.2, 0.0);\n"
531     "  vec3 black = vec3(0.0);\n"
532     "  colour = vec4(vec3[4](blue, white, red, black)[c], 1.0);\n"
533     "}\n"
```

```
534     ;
535     GLuint display = vertex_fragment_shader(s_vertex, s_fragment);
536     glUseProgram(display);
537     GLfloat m[4][4] = // ortho2d
538       { { 4.0 / W, 0, 0, 0 }
539       , { 0, 4.0 / H, 0, 0 }
540       , { 0, 0, -1, 0 }
541       , { -1, -1, 0, 1 }
542       };
543     GLint u_MVP = glGetUniformLocation(display, "MVP");
544     glUniformMatrix4fv(u_MVP, 1, GL_FALSE, &m[0][0]);
545     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
546     glEnableVertexAttribArray(0);
547     // prepare
548     init();
549     SF_INFO info = { 0, 48000, 2, SF_FORMAT_WAV | SF_FORMAT_FLOAT, 0, 0 };
550     SNDFILE *sndfile = 0;
551     unsigned char *video = 0;
552     if (RECORD)
553     {
554       sndfile = sf_open("meshwalk-2.0_a.wav", SFM_WRITE, &info);
555       video = malloc(w * h * 3);
556       // write 3 seconds silence for title
557       float frames[SR/FPS][2];
558       for (int i = 0; i < SR/FPS; ++i)
559         for (int c = 0; c < 2; ++c)
560           frames[i][c] = 0;
561       for (int k = 0; k < 3 * FPS; ++k)
562         sf_writef_float(sndfile, &frames[0][0], SR/FPS);
563     }
564     else
565     {
566       if (!(client = jack_client_open("meshwalk", JackNoStartServer, 0))) {
567         fprintf (stderr, "jack server not running?\n");
568         return 1;
569       }
570       jack_set_process_callback(client, processcb, 0);
571       /* create ports */
572       port[0] = jack_port_register(
573         client, "output_1", JACK_DEFAULT_AUDIO_TYPE, JackPortIsOutput, 0
574       );
575       port[1] = jack_port_register(
576         client, "output_2", JACK_DEFAULT_AUDIO_TYPE, JackPortIsOutput, 0
577       );
578       /* activate audio */
579       if (jack_activate(client)) {
580         fprintf (stderr, "cannot activate JACK client");
581         return 1;
582       }
583       /* must be activated before connecting JACK ports */
584       const char **ports;
585       if ((ports = jack_get_ports(
586         client, NULL, NULL, JackPortIsPhysical | JackPortIsInput
587       ))) {
588         /* connect up to two physical playback ports */
589         int i = 0;
590         while (ports[i] && i < 2) {
591           if (jack_connect(
592             client, jack_port_name(port[i]), ports[i]
593           )) {
594             fprintf(stderr, "cannot connect output port\n");
```

```
595           }
596           i++;
597         }
598         free(ports);
599       }
600     }
601     // main loop
602     glfwPollEvents();
603     for (int k = 0; !glfwWindowShouldClose(window); ++k)
604     {
605       // step
606       even();
607       odd(k);
608       if (k > 0x3FFFF)
609       {
610         tris();
611         glBindFramebuffer(GL_FRAMEBUFFER, fbo);
612         glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(t), t);
613         glClear(GL_COLOR_BUFFER_BIT);
614         glDrawArrays(GL_TRIANGLES, 0, (H/2 + 2) * (W/2 + 2) * 2 * 2 * 3);
615         glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
616         glBlitFramebuffer
617           ( 0, 0, w, h
618           , 0, 0, w, h
619           , GL_COLOR_BUFFER_BIT, GL_LINEAR
620           );
621         if (RECORD)
622         {
623           float audio[2][SR/FPS];
624           audiocb(&audio[0][0], &audio[1][0], SR/FPS);
625           float frames[SR/FPS][2];
626           for (int i = 0; i < SR/FPS; ++i)
627             for (int c = 0; c < 2; ++c)
628               frames[i][c] = audio[c][i];
629           sf_writef_float(sndfile, &frames[0][0], SR/FPS);
630           // download output frame
631           glBindFramebuffer(GL_READ_FRAMEBUFFER, 0);
632           glReadPixels(0, 0, w, h, GL_RGB, GL_UNSIGNED_BYTE, video);
633           // output PPM to stdout (flipped vertically)
634           printf("P6\n%d %d\n255\n", w, h);
635           fwrite(video, w * h * 3, 1, stdout);
636         }
637         // redisplay
638         glfwSwapBuffers(window);
639         glfwPollEvents();
640         if (RECORD && k >= 0x3FFFF + (4 * 60 - 6) * FPS)
641           break;
642         GLuint e;
643         while ((e = glGetError())) fprintf(stderr, "GL ERROR %d\n", e);
644       }
645     }
646     // cleanup
647     if (RECORD)
648     {
649       // write 3 seconds silence for credits
650       float frames[SR/FPS][2];
651       for (int i = 0; i < SR/FPS; ++i)
652         for (int c = 0; c < 2; ++c)
653           frames[i][c] = 0;
654       for (int k = 0; k < 3 * FPS; ++k)
655         sf_writef_float(sndfile, &frames[0][0], SR/FPS);
```

```
656         sf_close(sndfile);
657         free(video);
658     }
659     else
660     {
661         jack_client_close(client);
662     }
663     glfwDestroyWindow(window);
664     glfwTerminate();
665     return 0;
666 }
```